BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

First Semester 2018-19

Principles of Programming Languages

Lab Session 1: Functional Programming in Scala

What is Scala and Why Scala?

Scala is an object-oriented and functional programming language that runs on the Java Virtual Machine. It supports object oriented, functional and imperative programming paradigms. Scala has strong static type checking. Even though the compiler will infer the types of your variables, it's strongly typed in the same sense that Java is. That is, the compiler won't let you pass an object as a parameter to a method that requires an incompatible type, and it also won't let you call a method on an object if the object's type doesn't support that method.

For example:

If I declare a function add like this

```
def add(x:Double,y:Int):Double={x+y} //function returns a Double value
```

And call it like this

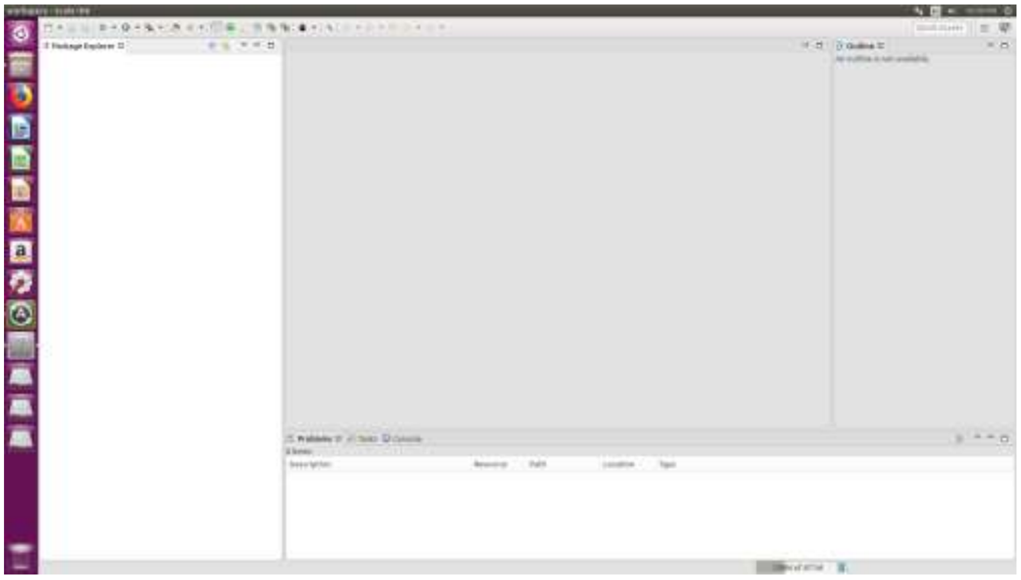
```
add(2.5,3.5)// The program will not compile and gives the following error:

error: type mismatch; found : Double(3.5) required: Int add(2.5,3.5)
```

This means the checking of data types is done at compile time. Scala source files are saved with .scala or .sc extension. Scala can be used for web application, desktop based application etc.

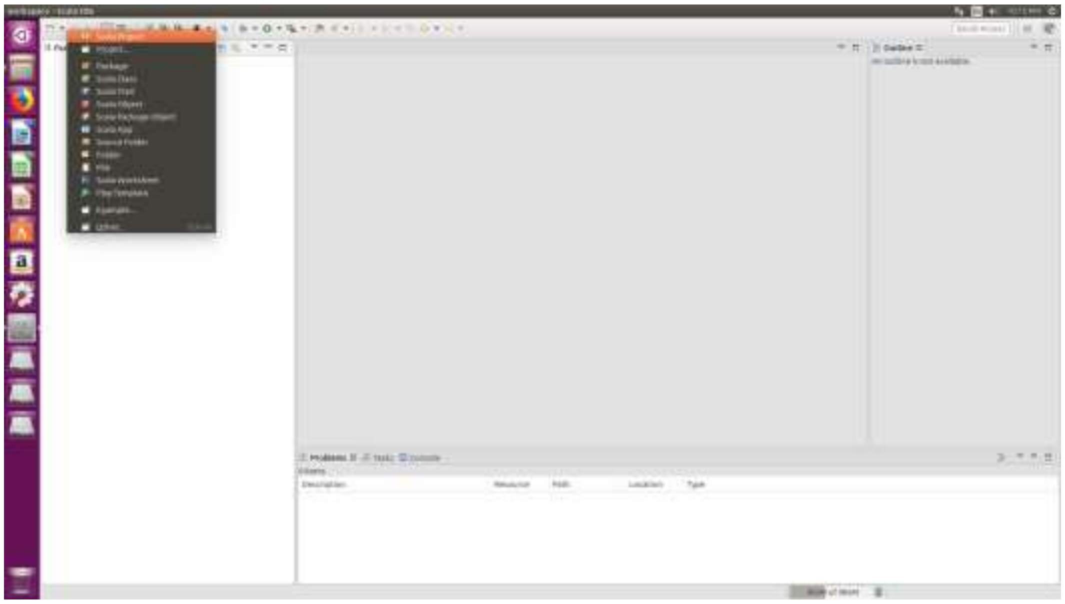
The name “Scala” comes not so creatively from the fact that the language can be used for “scalable” applications – applications that can grow with the demand of users. Immutable variables make it ideal for concurrent and parallel applications.

Setting up Scala

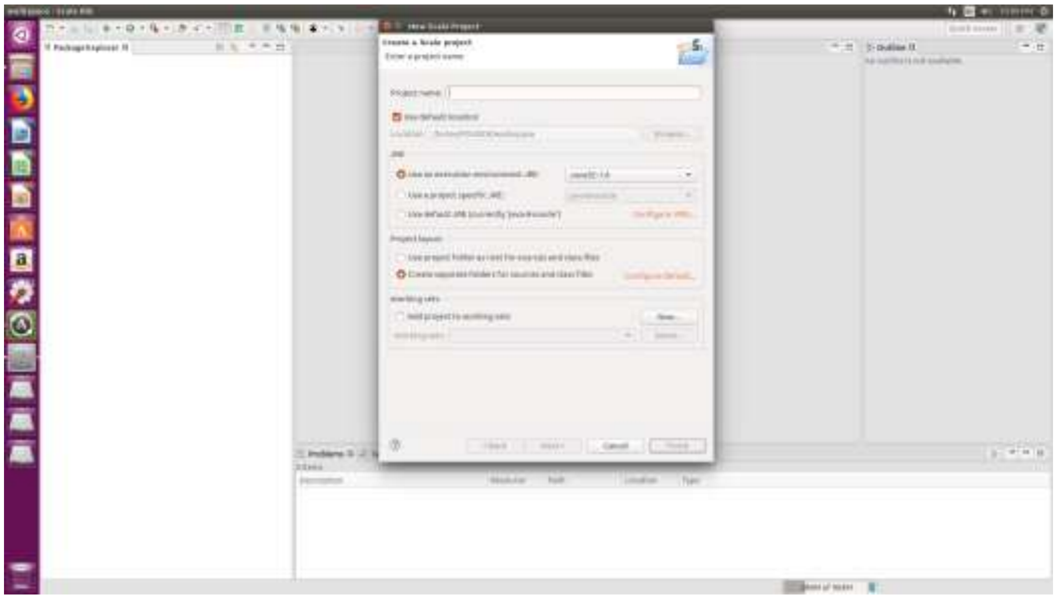
i) Open Eclipse from the desktop as shown.



ii) Click on **create new scala project** as shown below.



iii) Give a suitable name and click Finish.



Project name:

Use default location

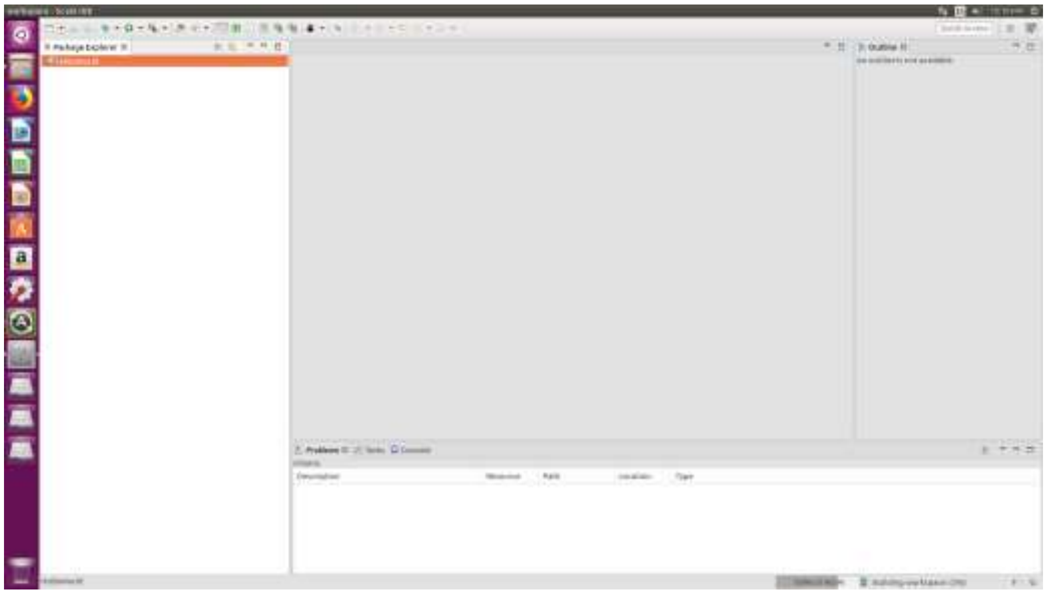
JRE

Project layout

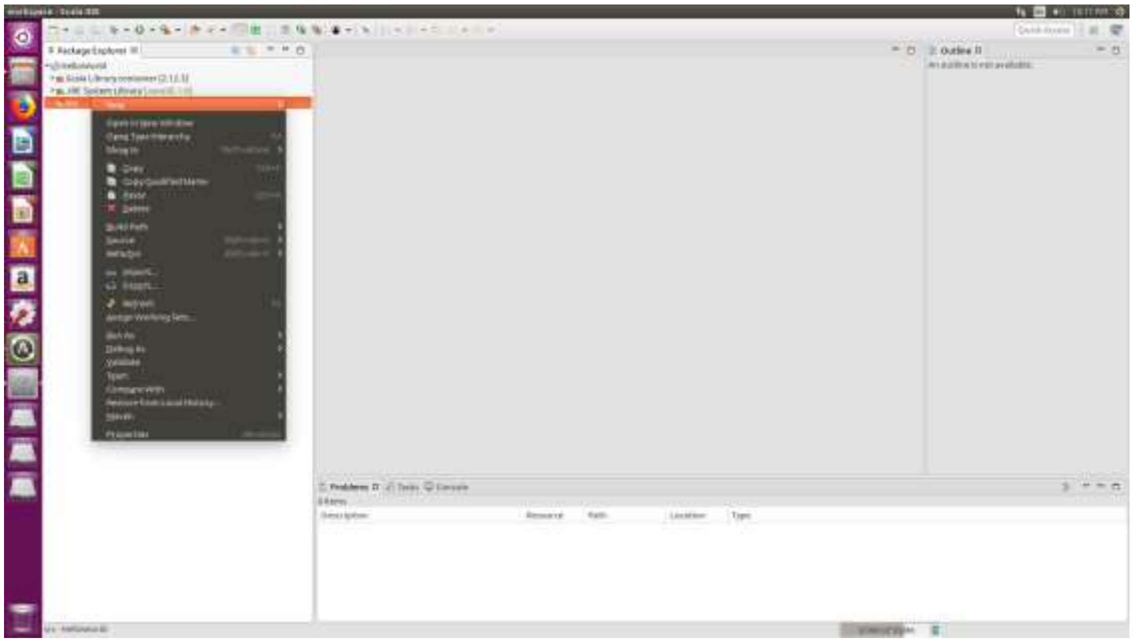
Create separate folders for sources and class files

Working sets

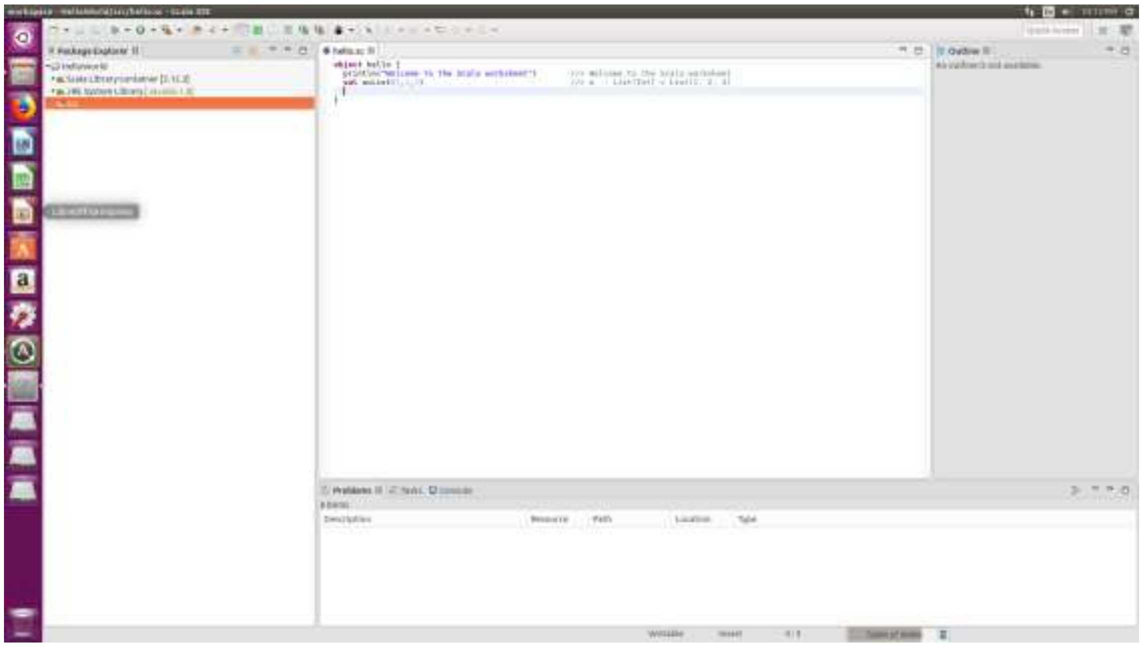
iv) The project with the name will be created as shown .



v) We need to create a new Scala worksheet here(.sc will be the extension) . Worksheet will interpret each line of scala code that is typed in to it and will be useful for seeing the results of the code as it is typed.



vi) Scala worksheet will be created. The inline comments refer to the output of the line.



```
object hello {
  println("Welcome to the Scala worksheet")       //> Welcome to the Scala worksheet
  val a=List(1,2,3)                               //> a  : List[Int] = List(1, 2, 3)
}
```

Hello World in Scala:

1. The Object Oriented way:

```
object Example{
def main(args:Array[String]){
    println "Hello World" //will output Hello World on the console
}
```

Note how it looks similar to a java Hello World program except for the object keyword instead of class. We will not be discussing about objects in Scala anymore as focus will be on functional programming paradigm.

2. The Functional way:

```
def Example{
    println("Hello World")
 }
 Example // this is the function call and it prints Hello World
```

Variables and Data types in Scala:

One can create mutable and immutable variable in Scala. Mutable variables are declared using the keyword “var” and immutable variables are declared using the “val” keyword. Consider the below example:

```
var a = 100
a = 101 // It works, No error.
val a = 100
a = 101// error: reassignment to val
```

Tip: Using var in Scala is discouraged since it goes against pure FP design

There are no primitive data types. Every type is an object, unlike in Java where we have int, float, etc.

Data type Name - Data type value

Byte - 8-bit signed two's complement integer

Short - 16-bit signed two's complement integer

Int - 32-bit two's complement integer

Long - 64-bit two's complement integer

Float - 32-bit IEEE 754 single-precision float

(positive or negative)

Double - 64-bit IEEE 754 double-precision float

(positive or negative)

Char - 16-bit unsigned Unicode character (0 to 2^16-1, inclusive)

String - a sequence of Chars

Boolean - true or false

Q) What is the 'Any' data type in Scala?? (Wait, don't skip over this question. You will require this for your assignment)

Lists

While functional programming in Scala, lists will be your go-to data structure.

It's built from "cons"(::) cells and ends in a Nil element.

You can create a List like this:

- val x = List(1, 2, 3)

or like this, using cons cells and a Nil element:

- val y = 1 :: 2 :: 3 :: Nil

Basic Operations on Lists

All operations on lists can be expressed in terms of the following three methods.

Sr.No	Methods & Description

1	Head This method returns the first element of a list.
2	tail This method returns a list consisting of all elements except the first.
3	isEmpty This method returns true if the list is empty otherwise false.

The following example shows how to use the above methods.

Example

```
object Demo {
   def main(args: Array[String]) {
      val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))
      val nums = Nil

      println( "Head of fruit : " + fruit.head )
      println( "Tail of fruit : " + fruit.tail )
      println( "Check if fruit is empty : " + fruit.isEmpty )
      println( "Check if nums is empty : " + nums.isEmpty )
   }
}
```

Output

```
Head of fruit : apples
Tail of fruit : List(oranges, pears)
Check if fruit is empty : false
Check if nums is empty : true
```

Concatenating Lists

You can use either ::: operator or <Listname>.:::() method or List.concat() method to add two or more lists. <> is a placeholder .Please find the following example given below −

Example

```
object Demo {
   def main(args: Array[String]) {
      val fruit1 = "apples" :: ("oranges" :: ("pears" :: Nil))
```

```
    val fruit2 = "mangoes" :: ("banana" :: Nil)
    // use two or more lists with ::: operator
    var fruit = fruit1 ::: fruit2
    println( "fruit1 ::: fruit2 : " + fruit )
    // use two lists with <Listname>.:::() method
    fruit = fruit1.:::(fruit2)
    println( "fruit1.:::(fruit2) : " + fruit )
    // pass two or more lists as arguments
    fruit = List.concat(fruit1, fruit2)
    println( "List.concat(fruit1, fruit2) : " + fruit  )
  }
}
```

Output

```
fruit1 ::: fruit2 : List(apples, oranges, pears, mangoes, banana)
fruit1.:::(fruit2) : List(mangoes, banana, apples, oranges, pears)
List.concat(fruit1, fruit2) : List(apples, oranges, pears, mangoes, banana)
```

Conditional expressions:

This is same as in other programming languages. Scala provides if,if-else and if-else ladder for the programmer's usage. Ternary Operator(?:) is not available in Scala. Example:

1. Simple if-else:

```
var age:Int = 19 //statically assign the type
if(age > 18){
        println ("Adult") } else{
        println ("Not an Adult");
}
```

2. If-else ladder:

```
var age:Int = 19;
if(age > 18 && age < 50){
        println ("Adult below 50");
} else if(age >50){
        println ("Adult above 50");
 } else{
```

```
                    println ("Not an Adult");
          }
```

Pattern Matching:

Scala's equivalent of a switch statement in other programming languages. They assist a lot while working with lists.

Example:

```
val value = “A” value match{
          case “A” => println("A")
          case “B” => println("B")
          case _ => println("None of the above")//default case
}
```

Functions:

Scala is a functional programming language, so functions form a very important part (duh!). Functions are first class values which means you can store function value, pass function as an argument and return function as a value from another function. We use def keyword to declare a function in scala. Data type of parameters needs to be specified without which the compiler will complain. Return type of the function is optional. If unspecified, compiler interprets it as Unit (equivalent to void in other programming languages) or the data type of the last statement in the function body will be considered as the return type.

```
          def funfunfunction() = { // Defining a function
            println("Welcome to funfunfunction")
            }
funfunfunction: ()Unit
scala> funfunfunction()
Welcome to funfunfunction //output
```

As mentioned, compiler will interpret the return type of this function as Unit as the last statement is println.

```
scala> def funfunfunction() {var a=5; println("Hey"); a} // Defining a function
funfunfunction: ()Unit
scala> def funfunfunction() = {var a=5; println("Hey"); a} // Defining a function
funfunfunction: ()Int
```

First, notice in the second example above that return type is Int because last statement has “a” and “a” has Int datatype. Secondly, notice how the compiler marks the first function as Unit while the other function is marked as Int. Notice the different function declarations. Reason: In Scala one can create function with or without = (equal) operator. If you use it, function will return value. If you don't use it, your function will not return anything and will work like subroutine.

How “complete” function declarations look like in Scala (i.e. with return type and parameter data types properly specified):

```
scala> def Add(a:Int, b:Int):Int = {  var c = a+b
           println(c)
           c //value of c will be returned from the function.
           }
          Add: (a: Int, b: Int)Int
scala> Add(2,3)
5
res12: Int = 5
```

<u>Example: Multiplication using Recursion</u>

```
def RecMul(a:Int, b:Int):Int = {
           if(b == 0) // Base condition
           0
           else
           a+RecMul(a,b-1)
           }
RecMul: (a: Int, b: Int)Int
scala> RecMul(2,3)
res13: Int = 6
```

Q) Can you implement Factorial using recursion?

Expression matching with list

When writing a recursive algorithm, you can take advantage of the fact that the last element in a List is a Nil object. For instance, in the following listToString method, if the current element is not Nil, the method is called recursively with the remainder of the List, but if the current element is Nil, the recursive calls are stopped and an empty String is returned, at which point the recursive calls unwind:

- def listToString(list: List[String]): String = list match {

```
    case s :: rest => s + " " + listToString(rest)
    case Nil => ""
}
```
OR

-
  ```
  def listToString(list: List[String]): String = list match {
      case :::(x,xs)  => x + " " + listToString(xs)
      case Nil => ""
  }
  ```

Cons is usually used instead of the '::' for breaking list into two parts, head and tail.

More Examples:

1) **Add a list**

```
def sum(ints: List[Int]): Int = ints match {
case Nil => 0
case :::(x,xs) => x + sum(xs)
}
```

2) **Optimized Product**

```
def product(ds: List[Double]): Double = ds match {
case Nil => 1.0
case :::(0.0, _) => 0.0
case :::(x,xs) => x * product(xs)
}
```

Functions with named parameters and default values:

Default values can be assigned to function parameters if needed. It helps in the scenario when you don't pass value during function calling. It will then use default values of parameters. In scala function, you can specify the names of parameters during calling the function,one can pass named parameters in any order or can also pass values only.

```
scala> def WeightedAdd(a:Int =2, b:Int=3):Int = {
        2*a+3*b
        }
WeightedAdd: (a: Int, b: Int)Int
scala> WeightedAdd() // will use the default values res17: Int = 13
scala> WeightedAdd(3,4)//this will use the parameters specified
```

res18: Int = 18

scala> WeightedAdd(b=3,a=4) //named paramters ; b will be taken as 3 and a as 4 ;order of parameters don't //matter res19: Int = 17

Some corner cases in named arguments (Calling with and without names) for reference:

scala> WeightedAdd(a=3,4) res22: Int = 18

scala> WeightedAdd(b=3,4) <console>:13: error: positional after named argument.

```
WeightedAdd(b=3,4)
                ^
```

scala> WeightedAdd(3,b=4) res24: Int = 18

Tail recursion in Scala

A tail-recursive function is just a function whose *very last action* is a call to itself. When you write your recursive function in this way, the Scala compiler can optimize the resulting JVM bytecode so that the function requires only one stack frame — as opposed to one stack frame for each level of recursion!

```
def sum(list: List[Int]): Int = list match {
  case Nil => 0
  case x :: xs => x + sum(xs)
}
```

Above example is not tail recursive as return needs to perform addition after recieving value of sum function. It can be converted as :

```
def sum(list: List[Int]): Int = {
    @tailrec
    def sumWithAccumulator(list: List[Int], currentSum: Int): Int = {
      list match {
        case Nil => {
          val stackTraceAsArray = Thread.currentThread.getStackTrace
          stackTraceAsArray.foreach(println)
          currentSum
        }
        case x :: xs => sumWithAccumulator(xs, currentSum + x)
      }
    }
    sumWithAccumulator(list, 0)
  }
```

Exercises (Please try on your own)

1) Fibonnaci series (Recursive)
2) Mean of List
3) GCD of two numbers
4) Reverse a list
5) Remove first n elements from a list